MethodsX 6 (2019) 1521–1542

Contents lists available at ScienceDirect

MethodsX

journal homepage: www.elsevier.com/locate/mex

Method Article

The Matlab code of the method based on the Full Range Factor for assessing the safety of masonry arches

Stefano Galassi*, Giacomo Tempesta

Department of Architecture, University of Florence, Piazza Brunelleschi 6, 50121 Florence, Italy

A B S T R A C T

In most masonry arches stresses are very low and, therefore, collapse does not occur because of material failure. As a consequence, the safety of arches should not be assessed by means of a safety factor based on material strength as for conventional structures. In 1969 Heyman was the first to state that the safety of masonry arches relies on their geometry and proposed a method for computing the so-called "geometrical factor of safety" based on the comparison between the shape of the thrust line and the profile of the arch. In this context, we have recently developed a method capable of computing the line of thrust closest to the geometrical axis and defining a safety factor based on the comparison between such a line of thrust and the profile of the arch, which we have denoted as "performance factor". In this paper, that supplements the author ref. (Tempesta and Galassi, 2019 [41]), the Matlab code of our method is provided for unlimited and unrestricted use by researchers as well as academics for educational purposes.

- The method (denoted as FRS Method) is inspired by the method proposed by Heyman in 1969
- Unlike the original iterative method, the FRS Method computes the line of thrust using a one-step procedure, which is less time consuming and provides the exact solution
- The original geometrical factor of safety is replaced by a performance factor, that characterizes the range of the equilibrium thrust lines within the profile of the arch effectively and the safety factor in a targeted way

A R T I C L E I N F O

Method name: Full Range Factor of Safety Method (FRS Method)

Keywords: Discrete arches, Line of thrust, Finite differences, Safety assessment, Heyman, Geometrical factor, Performance factor, Matlab code

Article history: Received 14 March 2019; Accepted 30 May 2019; Available online 4 June 2019

* Corresponding author.
E-mail address: stefano.galassi@unifi.it (S. Galassi).

https://doi.org/10.1016/j.mex.2019.05.033

Specifications Table

Subject Area:	*Engineering*
More specific subject area:	*Safety assessment of masonry arches based upon their geometry*
Method name:	*Full Range factor of Safety Method (FRS Method)*
Name and reference of original method	*The original method by Heyman* [1] *computes the line of thrust closest to the geometrical axis of an arch using an iterative method that proceeds by trial and error. The safety of the structure is successively assessed by defining the thickness of an ideal arch (i.e. the arch of minimal thickness) within the profile of the real arch (the geometrical factor of safety).* [1] *J. Heyman, The safety of masonry arches, Int. J. Mech. Sci. (1969) 11(4): 363-85.*
Resource availability	[41] *G. Tempesta, S. Galassi, Safety evaluation of masonry arches. A numerical procedure based on the thrust line closest to the geometrical axis, Int. J. Mech. Sci. (2019) 155: 206–21.*

Method details

Background

Collapses of masonry arches which occurred in past times clearly demonstrated that their geometrical profile is the main feature responsible for safety. Based on this, Heyman [1] proposed an iterative method for computing the line of thrust closest to the geometrical axis and pointed a factor for assessing the safety, which relied exclusively on geometrical considerations. Such a factor, denoted as “geometrical factor of safety”, was defined as the ratio between the actual thickness of the arch and the thickness of the minimal arch within the profile of the real one, obtained by scaling its thickness to enclose the thrust line. Thenceforth, the scientific community has shared the Heymanian school of thinking and his theory has been used to assess the safety of masonry arches and vaults in the context of limit or incremental analysis [2–28].

Limit analysis is used to assess the load-carrying capacity and the safety level, avoiding the mechanical properties of materials to be estimated [29,30] as is required by non-linear elastic incremental analyses [31–37], through which the “exact” solution and the “true” line of thrust can be achieved, but long iterative analyses are needed. In the literature, methodologies and computer programs based on the kinematic theorem [5,14,17], denoted as “mechanism methods”, or on the static theorem [3,4,6,15], denoted as “thrust line methods”, have been proposed and have largely replaced the earlier hand based techniques, such as the famous Mery’s method [38].

Generally, mechanism methods assume that a masonry arch becomes a mechanism when at least four hinges occur. However, hinge position is unknown and procedures based on these methods must assume trial positions and perform several computations, using the equilibrium equations at the hinges [5] or the equations of virtual work [14]. Since the theorems of limit analysis do not provide unique solutions for the collapse load factor if a non-associative friction rule is assumed [39], in order to also take into account sliding mechanisms due to finite friction, robust numerical procedures have been formulated [6,18,19,25], but they usually use a static equilibrium approach.

Thrust line methods, instead, assess the safety level using procedures that compute the line of thrust by solving the equilibrium equations or a linear programming problem and identify the zone where the inner forces (i.e. the thrust line) can stand. This zone is a domain that has been defined in the literature according to the middle third rule, which is derived from the elastic theory, or the middle half rule, which is a less conservative approach. Lastly, Heyman [1] proposed a revisited version of the aforementioned rules considering that the thrust line can lie within the whole thickness of the arch and, therefore, removed the boundaries of the middle third rule or the half middle rule of the thickness. This criterion is the less conservative one and, indeed, it corresponds to the limit equilibrium condition of an arch and has also been extended to the analysis of vaults [26–28]. All these variants of the thrust line method can be summarized [40] by the Heymanian concept of geometric factor of safety mentioned above, hereafter referred to as the “GFS Method”. In order to overcome limitations of the Heymanian geometric factor of safety for

practical use in case of irregular profiles, in [20] the domain of safety has been redesigned as the locus of admissible positions of poles in the force diagrams leading to thrust lines that lie entirely within the masonry envelope and a safety indicator has been pointed out based on the value of the maximum and minimum admissible thrusts.

The authors of this paper, instead, have developed a new method (denoted as "Full Range factor of Safety-based Method", hereafter referred to as the "FRS Method") that improves the original method proposed by Heyman in 1969. The FRS Method is described in detail in [41] and this paper, which provides a brief technical description of it in the next two sections and the full Matlab code in the following pages, supplements author's ref. [41]. Readers can use and modify these Matlab routines without restrictions for research and educational purposes. Conversely, professional use of this code is forbidden and the authors will not be responsible for it.

Unlike the original method proposed by Heyman, that is an iterative method that computes the line of thrust closest to the geometrical axis proceeding by trial and errors, the FRS Method is a one-step procedure. Thus, this new method allows obtaining the "exact" solution of the problem and a reduction in time is assured. Furthermore, a new domain of safety has been conceived and obtained by shifting the thrust line vertically until it touches the extrados and intrados profiles of the arch. This domain of safety, that has been denoted as "full-range of equilibrium thrust lines", provides a safety indicator (the performance factor) computed as the ratio between the vertical thickness of the domain and the minimum vertical thickness of the arch.

This procedure applies to any profile of the arch, subject only to vertical forces. In [41] the inverted catenary-shaped arch, the circular arch, the segmental arch, the pointed arch, the rampant arch and also an arch of generic shape have been analyzed. In the current formulation, the procedure cannot analyze multi-ring nor multi-span arches. At present, we are developing a new release of the method in order to also account for the horizontal actions provoked by an earthquake and the extension of the method will be capable of assessing the safety of 3D-structures such as barrel and cross vaults.

Brief description of the method

The arch is regarded as a continuous structure but, in order to allow the method to be generalized to all geometrical profiles and load conditions, it is conveniently divided into a finite number n of

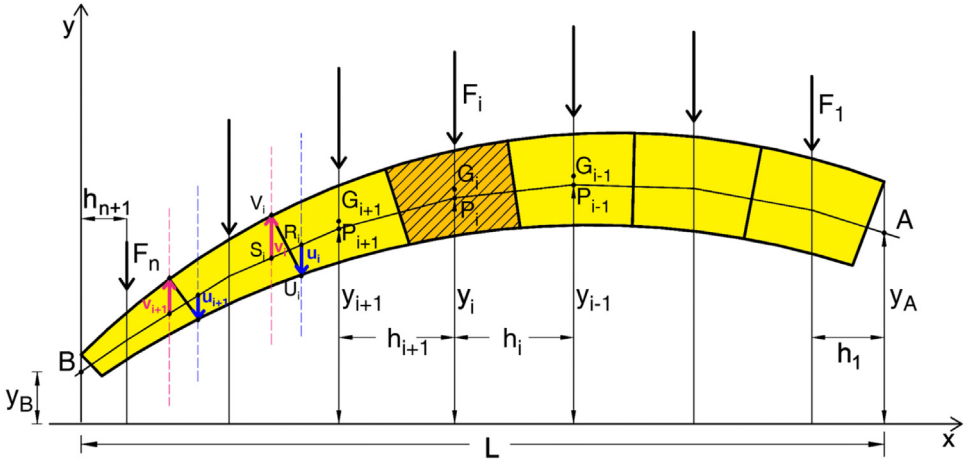

Fig. 1. The arch discrete model (adapted from [41]).

discrete elements (Fig. 1). In so doing, the analysis of a rigid block arch is also allowed, matching the blocks with the elements and the joints between blocks with the lines of separation between them.

Hereafter, the main steps of the method are listed, although for a more in-depth understanding of the numerical procedure we suggest that the interested reader examines the main paper [41]:

1 For each discrete element *i* of the arch the position of the centroid G_i and the vertical load F_i acting on it are computed;
2 In order to identify a specific line of thrust (that closest to the geometrical axis, as conceived in [42]) among the ∞^3 likely lines of thrust in equilibrium with the loads, three parameters or conditions have been imposed: the ordinate Y_0 of point A and the ordinate Y_{n+1} of point B, for which the first and the last segments of the thrust line must pass, and the horizontal component of the thrust H;
3 To ensure that the line of thrust identified at the previous step is exactly the one closest to the geometrical axis, the distances $d_i = Y_i - Y_{Gi}$ among the vertices P_i of the line of thrust and the centroids G_i that define the geometrical axis are minimized.

The minimization procedure is written in algebraic form and makes use of the finite difference method. Therefore, the system of the equilibrium equations of the internal and external forces acting on each vertex of the line of thrust is written (Eq. (1)):

$$\begin{cases} \left(\dfrac{-1}{h_1}\right)\cdot Y_0 + \left(\dfrac{h_1+h_2}{h_1\cdot h_2}\right)\cdot Y_1 + \left(\dfrac{-1}{h_2}\right)\cdot Y_2 = \dfrac{F_1}{H} \\ \left(\dfrac{-1}{h_2}\right)\cdot Y_1 + \left(\dfrac{h_2+h_3}{h_2\cdot h_3}\right)\cdot Y_2 + \left(\dfrac{-1}{h_3}\right)\cdot Y_3 = \dfrac{F_2}{H} \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ \left(\dfrac{-1}{h_i}\right)\cdot Y_{i-1} + \left(\dfrac{h_i+h_{i+1}}{h_i\cdot h_{i+1}}\right)\cdot Y_i + \left(\dfrac{-1}{h_{i+1}}\right)\cdot Y_{i+1} = \dfrac{F_i}{H} \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ \left(\dfrac{-1}{h_n}\right)\cdot Y_{n-1} + \left(\dfrac{h_n+h_{n+1}}{h_n\cdot h_{n+1}}\right)\cdot Y_n + \left(\dfrac{-1}{h_{n+1}}\right)\cdot Y_{n+1} = \dfrac{F_n}{H} \end{cases} \tag{1}$$

The unknowns of the system in Eq. (1) are the *n+2* ordinates Y_i of vertices of the line of thrust and the thrust H. Therefore, it is indeterminate to three degrees and to solve it the three conditions mentioned at step 2 are imposed. For this purpose, putting $K = H^{-1}$, the system is rearranged isolating the three redundant unknowns (Eq. (2)):

$$\begin{bmatrix} \left(\dfrac{h_1+h_2}{h_1\cdot h_2}\right) & \left(\dfrac{-1}{h_2}\right) & 0 & 0 \\ \left(\dfrac{-1}{h_2}\right) & \left(\dfrac{h_2+h_3}{h_2\cdot h_3}\right) & \left(\dfrac{-1}{h_3}\right) & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & \left(\dfrac{-1}{h_i}\right) & \left(\dfrac{h_i+h_{i+1}}{h_i\cdot h_{i+1}}\right) & \left(\dfrac{-1}{h_{i+1}}\right) \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \left(\dfrac{-1}{h_n}\right) & \left(\dfrac{h_n+h_{n+1}}{h_n\cdot h_{n+1}}\right) \end{bmatrix} \cdot \begin{Bmatrix} Y_1 \\ Y_2 \\ \cdots \\ Y_i \\ \cdots \\ Y_n \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \\ \cdots \\ F_i \\ \cdots \\ F_n \end{Bmatrix} \cdot K + \begin{Bmatrix} 1/h_1 \\ 0 \\ \cdots \\ 0 \\ \cdots \\ 0 \end{Bmatrix} \cdot Y_0 + \begin{Bmatrix} 0 \\ 0 \\ \cdots \\ 0 \\ \cdots \\ 1/h_{n+1} \end{Bmatrix} \cdot Y_{n+1} \tag{2}$$

and written, more compactly, in matrix form (Eq. (3)):

$$[D]\{Y\} = \{T_1\}\cdot K + \{T_2\}\cdot Y_0 + \{T_3\}\cdot Y_{n+1} \tag{3}$$

The solution of Eq. (3), provided by Eq. (4), is not computable yet because K, Y_0, Y_{n+1} are unknown parameters:

$$\{Y\} = \{R_1\}\cdot K + \{R_2\}\cdot Y_0 + \{R_3\}\cdot Y_{n+1} \tag{4}$$

In Eq. (4), for clarity of reading, is put: $\{R_1\} = [D]^{-1}\{T_1\}$,$\{R_2\} = [D]^{-1}\{T_2\}$, $\{R_3\} = [D]^{-1}\{T_3\}$.

Thus, as the objective of the analysis is the detection of the line of thrust closest to the geometrical axis of the arch, in order to compute the value of the three unknown parameters the function S, which expresses the square of the distances between the vertices of the line of thrust and the centroids of the

elements, is minimized (Eq. (5)):

$$S = (\{R_1\} \cdot K + \{R_2\} \cdot Y_0 + \{R_3\} \cdot Y_{n+1} - \{Y_G\})^2 \tag{5}$$

To minimize function S, the conditions that express the zeroing of the three partial derivatives with respect to the three unknown parameters are formulated (Eq. (6)):

$$\begin{aligned} &\frac{\partial S}{\partial K}(Y_0,\ Y_{n+1}, K) = 0 \\ &\frac{\partial S}{\partial Y_0}(Y_0,\ Y_{n+1}, K) = 0 \\ &\frac{\partial S}{\partial Y_{n+1}}(Y_0,\ Y_{n+1}, K) = 0 \end{aligned} \tag{6}$$

Developing Eq. (6), we obtain the final system of three linear equations (Eq. (7)), whose solution provides the value of the three unknown parameters:

$$\begin{bmatrix} \{R_1\}^2 & \{R_1\}\{R_2\} & \{R_1\}\{R_3\} \\ \{R_1\}\{R_2\} & \{R_2\}^2 & \{R_2\}\{R_3\} \\ \{R_1\}\{R_3\} & \{R_2\}\{R_3\} & \{R_3\}^2 \end{bmatrix} \cdot \begin{Bmatrix} K \\ Y_0 \\ Y_{n+1} \end{Bmatrix} = \begin{Bmatrix} \{R_1\}\{Y_G\} \\ \{R_2\}\{Y_G\} \\ \{R_3\}\{Y_G\} \end{Bmatrix} \tag{7}$$

The ordinates of the vertices of the line of thrust are lastly computed substituting backwards such parameters into Eq. (4).

Full-range factor of safety

Finally, the line of thrust closest to the geometrical axis, obtained using the method above, is used to assess a factor of safety. According to Heyman [1], since arches have low stresses and collapse does not generally occur because of material failure, such a factor can be assessed based on their geometry by simply comparing the shape of the line of thrust to the profile of the arch. The safety factor is identified based on the line of thrust "capacity" of being moved vertically while still remaining contained within the profile of the arch. It is worth noting that, the arch being discretized in a finite number of elements (or blocks), the check of the line of thrust being contained within the profile of the arch must be carried out only in correspondence to the lines of separation among the elements (i.e. the joints of a rigid block), where the points of pressures (points of application of the inner resultant forces) are placed.

The line of thrust is vertically shifted, both upwards and downwards, until it becomes tangent to the extrados and intrados curves of the arch, in such a way to obtain two limit lines of thrust. These lines, denoted as "upper and lower bound thrust lines", identify a region that describes the domain of the admissible stress states, which is the domain of the admissible lines of thrust parallel to that provided by the procedure (domain of safety).

As reported in [42], to detect the lower (upper) bound of the domain a four step algorithm has been conceived (Fig. 1):

1) The points R_i (S_i) (with i = 1 to n+1) of intersection between the straight lines passing through all the intrados (extrados) point U_i (V_i) of the discrete arch and the thrust line are computed;
2) The vertical vectors $\mathbf{u_i} = (U_i - R_i)$ and $\mathbf{v_i} = (V_i - S_i)$ are determined;
3) The vertical distance to which the line of thrust must be shifted to obtain the lower bound of the domain is given by the maximum modulus among all vectors $\mathbf{u_i}$ and the vertical distance to which the line of thrust must be shifted to obtain the upper bound of the domain is given by the minimum modulus among all vectors $\mathbf{v_i}$;
4) The vectors $\{Y_{inf}\} = \{Y\} + \Delta Y_{inf}$ and $\{Y_{sup}\} = \{Y\} + \Delta Y_{sup}$ are lastly computed, whose entries are the ordinates of the points that define the lower and upper lines of thrust respectively, i.e. the lower and upper bound of the domain.

The ratio between the vertical thickness of the arch and the vertical thickness of the domain provides the “full-range factor of safety”, which is a reinterpretation of the “geometrical factor of safety” proposed by Heyman in [1]. Nevertheless, the analyst should assess the safety of an arch referring to the “performance factor”, computed as the reciprocal of the “full-range factor of safety”, because in so doing the range of the factors indicating the degree of safety is comprised between 0 (the lowest safety degree) and 1 (the highest safety degree). Negative factors point out unsafe arches, because they correspond to a negative thickness of the domain of equilibrium thrust lines.

Nomenclature of variables	
Brick	Struct array containing data of the discrete elements of the arch: ‘x(i)’,y’(i)’ are the coordinates of the i-th vertex of the closed polygon that defines the contour of each element (i = 1–4); ‘xG(i)’ and ‘yG(i)’ are the coordinates of the i-th element centroid; ‘Weight’ is the weight of the element and ‘Fy’ is the value of the likely additional vertical force in correspondence to an element.
MaxNumBrick	Is the number of discrete elements.
ResearchLoadFACTOR	If it is set equal to 1 then an additional force Fy in correspondence to an element is assumed to be inputted by the user and the program computes the collapse factor. Otherwise, the value can be set equal to 0 and the analysis is performed for the assigned self-weight loads.
VectorF	Entries of this vector are the horizontal force, vertical force and the moment acting in correspondence to the centroid of each element
OptShape	Struct array containing variables relative to the analysis. If ‘Type_Analysis’ is set by the user equal to 1 then the FRS method is used to compute the safety factor (i.e.: the performance factor); if it is set equal to 2 then the GFS is used (i.e.: the geometrical factor of safety). ‘LoadFactor’ is used by the program and is equal to 1 when only self-weights are present on the arch; increasing values (starting from 1) are used for computing the collapse factor due to an additional increasing load. Entries of vectors 'Vector_Y' and 'Vector_X' are the ordinates and abscissae of the vertices of the line of thrust respectively. Entries of vectors 'Vector_xCPdx', 'Vector_yCPdx', 'Vector_xCPsx' and 'Vector_yCPsx' are the absissae and ordinates of the intersection points between the line of thrust and the right and left joint of each element (i.e. the points of pressure). Entries of variables 'sSUP' and 'sINF' are the superior and inferior distances between the line of thrust and the extrados and intrados curves respectively. ‘sID’ is the overall thickness of the ideal arch (sSUP + sINF), measured along the joint orientation, when the GFS Method is set by the user. Instead, if the FRS Method is set, ‘sID’ measures the vertical thickness of the domain of equilibrium. ‘RealArchMinimumThickness’ is the minimum thickness of all joints. ‘Geom_Safety_Factor’ is the geometrical factor of safety (GFS approach) or the full-range factor of safety (FRS approach). Entries of 'Vector_Ysup' and 'Vector_Yinf' are the ordinates of the upper and lower bounds of the domain of the equilibrium thrust lines if the FRS Method is set by the user. Entry of variable ‘ThrustH’ is the value of the thrust.

Main routine of the Matlab code

The FRS Method was originally implemented in the software *ArchiVAULT* [43] written both in VisualBASIC 6.0 as for the graphical user interface and in VisualC++ 6.0 as for the numerical computations. *ArchiVAULT* is a complete computer program for performing the analysis of masonry arches developed by the authors. Therefore, with the purpose of providing a stand-alone program specifically dedicated to the assessment of the safety of arches through the FRS Method and freely usable by researchers and academics, routines regarding the FRS Method were translated in Matlab. In this section the Matlab code is provided and discussed. The code is organized in various routines and the routines are saved to separate files. The Matlab language does not distinguish between routines and functions, however, for the sake of clarity, hereafter routines that return a value will be referred to as “functions” and routines that only execute operations will be referred to as “routines”.

The main section of the code that follows must be stored in the “FRS_Method.m” file by the user. In the first lines the set of global variables available for all routines are defined. The nomenclature of the variables is listed above.

```
%*********************************************
%*                FRS Method                 *
%*                ----------                 *
%* Release: 1.0.0 - 2018-2019                *
%* Developed: Matlab v.6.1.0.450 Release 12.1  *
%* Authors: Stefano Galassi & Giacomo Tempesta *
%*********************************************

%Clear all variables from memory
clear all
%***********************************
%Global variables
global ResearchLoadFACTOR
global Brick
Brick = struct('x',0,'y',0,'xG',0,'yG',0,'Weight',0,'Fy',0)
global VectorF
global MaxNumBrick
global OptShape
OptShape=struct('Type_Analysis',2,'LoadFactor',1,'Vector_Y',0,'Vector_X',0,'Vector_xCPdx',0,'Vector_yCPdx',0,'Vector_xCPsx',0,'Vector_y
CPsx',0,'sSUP',0,'sINF',0,'sID',0,'RealArchMinimumThickness',0,'Geom_Safety_Factor',1,'Vector_Ysup',0,'Vector_Yinf',0,'ThrustH',0)
```

The lines that follow are devoted to the geometrical modeling of the arch, the weight of the elements and the type of analysis to be performed. The user must customize these lines to define a user-defined arch to be analyzed. The parameters set below refers to the Random Arch described in [41].

```
%***********************************
%User Input Data
filename='RandomArch.txt' %name of the input file
Depth=100; %depth of the arch (cm)
UnitWeight=2000; %unit weight of the material (kgf/m^3)
MaxNumBrick=7; %number of bricks
OptShape.Type_Analysis=1; %1=FRS our modified method - 2=GFS by Heyman
ResearchLoadFACTOR=0; %0=NO - 1=YES
%***********************************
```

Then the code reads the input TXT file that contains data regarding the shape of the arch and stores the coordinates of the four vertices of each element in the struct vectors Brick(. . .).x(. . .) and Brick (. . .).y(. . .). The TXT file must be prepared by the user. In the “Input data file” Section the way the user must prepare this file is explained.

```
fileID = fopen(filename,'r'); %load the geometry: name of the input file
%Lines beginning with % are not considered
[NumPunto,X,Y] = textread(filename,'%s%f%f','commentstyle','matlab');
fclose(fileID);

%Assign coordinates of element vertices to the variables
j=0;
for indexBrick = 1 : MaxNumBrick
    for i = 1 : 4
        j=j+1;
        Brick(indexBrick).x(i)=X(j);
        Brick(indexBrick).y(i)=Y(j);
        Brick(indexBrick).Fy=0; %(kgf)
    end
end
%Brick(3).Fy=-2000; %input an additional vertical force in correspondence to the third element (kgf)
```

Lines that follow call the ‘ComputeElementCentroid’ routine, that computes the coordinates of the centroid of each element.

```
 %Compute the coordinates of element centroids
 for indexBrick = 1 : MaxNumBrick

           [Brick(indexBrick).xG,Brick(indexBrick).yG,Brick(indexBrick).Weight]=ComputeElementCentroid
       (Brick(indexBrick).x,Brick(indexBrick).y,Depth,UnitWeight);
end
```

Then the procedure calls the ‘RunAnalysis’ routine, that computes the line of thrust closest to the geometrical axis and, successively, performs the safety assessment through the FRS Method proposed by the authors or through the geometrical factor based method by Heyman (GFS).

```
%Compute the line of thrust closest to the geometrical axis and the domain of equilibrium (GFS o FRS)
RunAnalysis
```

The last part of the code creates a window where the discrete arch is plotted. The line of thrust and the upper and lower bounds of the ideal arch (Heymanian approach) or of the domain of equilibrium thrust lines (our approach) is also plotted on it. The values of the safety factor and the thickness of the ideal arch are also presented in the legend.

```
%************
%* GRAPHICS *
%************
axis equal
grid on
xlabel('abscissae [cm]')
ylabel('ordinates [cm]')

%plot the elements of the arch
for indexBrick = 1 : MaxNumBrick
    hold on
    plot(Brick(indexBrick).x,Brick(indexBrick).y,'b') %border of elements (blue)
    fill(Brick(indexBrick).x,Brick(indexBrick).y,'y') %fill of the elements (yellow)
    hold on
    scatter(Brick(indexBrick).xG,Brick(indexBrick).yG,2,'b') %element centroid (blue)
    hold on
    elementID = int2str(indexBrick);
    text(Brick(indexBrick).xG,Brick(indexBrick).yG,elementID) %write the element number
end
%plot the line of thrust closest to the geometrical axis
hold on
ThrustLine=plot(OptShape.Vector_X,OptShape.Vector_Y,'r','linewidth',2); %red and thick thrust line
%plot the points of pressure in the right and left interfaces
hold on
scatter(OptShape.Vector_xCPdx, OptShape.Vector_yCPdx,3,'r') %points of pressure in the right interface (red)
hold on
scatter(OptShape.Vector_xCPsx, OptShape.Vector_yCPsx,3,'r') % points of pressure in the left interface (red)

%plot the safety domain (GFS or FRS)
if OptShape.Type_Analysis == 1 %"FullRangeFactor"
      %Plot the two limit lines of thrust (tangent to the intrados and extrados)
      %l.o.t. tangent to the intrados
    Lbound=plot(OptShape.Vector_X,OptShape.Vector_Yinf,'b'); %upper bound (blue)
      % l.o.t. tangent to the extrados
    Ubound=plot(OptShape.Vector_X,OptShape.Vector_Ysup,'g'); %lower bound (green)
      %Title and Legend
    title('Full Range Factor of Safety')
    legend([ThrustLine,Ubound,Lbound],'Thrust Line','Upper bound','Lower bound')
    IdealArchThickness=strcat('Ideal Arch Thickness: ',num2str(OptShape.sID,'%.2f'),' cm');
    text(min(xlim)+10,min(ylim)+130, IdealArchThickness)
    FullRangeFactor=strcat('Full Range Factor of Safety (k): ',num2str(OptShape.Geom_Safety_Factor,'%.4f'));
    text(min(xlim)+10,min(ylim)+100, FullRangeFactor)
    PerformanceFactor=strcat('Performance Factor (1/k): ',num2str(1 / OptShape.Geom_Safety_Factor,'%.4f'));
    text(min(xlim)+10,min(ylim)+70, PerformanceFactor)
    if OptShape.sID < 0
        text(min(xlim)+10,min(ylim)+40, 'No thrust line of this shape lies within the arch profile! Unsafe arch!')
    end

else % 2 = "GeometrialFactorOfSafety"
      %Plot the ideal arch within the real one, reducing the thickness of the arch.
      %define the intrados and extrados curves of the arch
    for i = 1 : MaxNumBrick
          angleDX = OrientedLineAngle(Brick(i).x(2) - Brick(i).x(1), Brick(i).y(2) - Brick(i).y(1)); %right angle
          angleSX = OrientedLineAngle(Brick(i).x(3) - Brick(i).x(4), Brick(i).y(3) - Brick(i).y(4)); %left angle
          j=i+1;
          UboundX(i) = (Brick(i).x(1) + Brick(i).x(2)) / 2 + OptShape.sSUP * cos(angleDX);
          UboundX(j) = (Brick(i).x(3) + Brick(i).x(4)) / 2 + OptShape.sSUP * cos(angleSX);
          UboundY(i) = (Brick(i).y(1) + Brick(i).y(2)) / 2 + OptShape.sSUP * sin(angleDX);
          UboundY(j) = (Brick(i).y(3) + Brick(i).y(4)) / 2 + OptShape.sSUP * sin(angleSX);

          LboundX(i) = (Brick(i).x(1) + Brick(i).x(2)) / 2 - OptShape.sINF * cos(angleDX);
          LboundX(j) = (Brick(i).x(3) + Brick(i).x(4)) / 2 - OptShape.sINF * cos(angleSX);
          LboundY(i) = (Brick(i).y(1) + Brick(i).y(2)) / 2 - OptShape.sINF * sin(angleDX);
          LboundY(j) = (Brick(i).y(3) + Brick(i).y(4)) / 2 - OptShape.sINF * sin(angleSX);
    end
          Ubound=plot(UboundX,UboundY,'g'); %upper bound
          Lbound=plot(LboundX,LboundY,'b'); %lower bound
            %Title and Legend
          title('Geometrical Factor of Safety')
          legend([ThrustLine,Ubound,Lbound],'Thrust Line','Upper bound','Lower bound')
          IdealArchThickness=strcat('Ideal Arch Thickness: ',num2str(OptShape.sID,'%.2f'),' cm');
          text(min(xlim)+10,min(ylim)+130, IdealArchThickness)
          GeometricalFactor=strcat('Geometrical Factor of Safety (k): ',num2str(OptShape.Geom_Safety_Factor,'%.4f'));
          text(min(xlim)+10,min(ylim)+100, GeometricalFactor)
          if OptShape.sID > OptShape.RealArchMinimumThickness
              text(min(xlim)+10,min(ylim)+70, 'No thrust line of this shape lies within the arch profile! Unsafe arch!')
          end
      end
      if ResearchLoadFACTOR == 1  %YES
          Message=strcat('Collapse factor: ',num2str(OptShape.LoadFactor,'%.4f'));
          text(min(xlim)+10,min(ylim)+10, Message)
      end
```

Input data file

By means of a text editor such as NotePad, the user must define a text file (*.txt) for inputting the geometry of the arch in the Matlab program. Fig. 2a shows the text file of the Random Arch, that is a structure subdivided into seven elements. Elements are numbered from right to left and each element is defined as a closed polygon composed of four vertices. Vertices 1 and 2 define the right joint; vertices 3 and 4 define the left joint. Points 1 and 4 are at the intrados of the element and points 2 and 3 are at the extrados. The structure of the text shown if Fig. 2a must be used, in which lines beginning with the '%' character are not read by the Matlab program.

The "ComputeElementCentroid" function

This function computes the coordinates of element centroids. It must be saved to the 'ComputeElementCentroid.m' file by the user. The main routine passes to this function the coordinates of the four vertices of each element, the depth of the arch and the unit weight of the material. Exploiting the Varignon's theorem, this function returns the element centroid ('xG', 'yG') and the weight of the element ('Weight'), by means of a procedure that computes the first moment of area about the x and y axis and the area of the element.

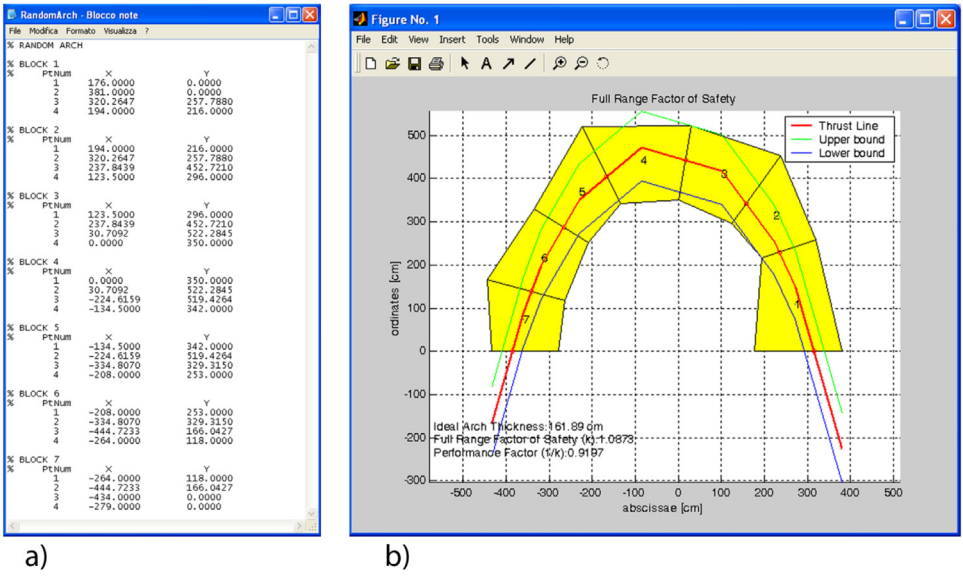

Fig. 2. a) Input data file of the Random Arch described in [41]; b) output of results provided by this Matlab code.

```
function [xG,yG,Weight] = ComputeElementCentroid(X,Y,Depth,UnitWeight)

%************
%*   Area   *
%************

Area = 0;
for i = 1 : 4
    J = i + 1;
    if i == 4
        J = 1;
    end
    Area = Area + 0.5 * (Y(i) + Y(J)) * (X(J) - X(i));
end

%************************
%* First Moment of Area *
%* about the x axis     *
%************************

Sx = 0;
for i = 1 : 4
    J = i + 1;
    if i == 4
        J = 1;
    end
    Sx = Sx + (1 / 6) * (Y(i) ^ 2 + Y(i) * Y(J) + Y(J) ^ 2) * (X(J) - X(i));
end

%************************
%* First Moment of Area *
%* about the y axis     *
%************************

Sy = 0;
for i = 1 : 4
    J = i + 1;
    if i == 4
        J = 1;
    end
    Sy = Sy + (-1 / 6) * (X(i) ^ 2 + X(i) * X(J) + X(J) ^ 2) * (Y(J) - Y(i));
end

%********************
%* Centroid G(x,y) *
%********************

xG = Sy / Area;
yG = Sx / Area;

%*************************
%* Weight of the element *
%*************************
Weight = -Area * Depth * UnitWeight / 1000000;


return
```

The “RunAnalysis” routine

This routine, that must be saved to the ‘RunAnalysis.m’ file, runs the analysis and the line of thrust closest to the geometrical axis is computed by calling the BestThrustLine(TempLoadFACTOR) routine. Then, based on the method chosen by the user (FRS or GFS method), the safety verification is performed. If variable OptShape.Type_Analysis is set equal to 1, then the FullRangeFactor routine is called and the domain of equilibrium thrust lines is computed (FRS Method). Instead, if variable OptShape.Type_Analysis is set equal to 2, then the GeomFactor routine is called and the ideal arch is defined (GFS Method). Additionally, this routine also checks if the load factor is required to be computed (this occurs when the TempLoadFACTOR variable has been set equal to 1 by the user). If the load factor is required to be computed, the TempLoadFACTOR variable assumes values greater than

1 and the analysis is rerun iteratively until the limit condition of equilibrium is attained. As an example, Fig. 2b shows the graphical output of the results of the Random Arch, subject to its self-weight, provided by this program.

```
function RunAnalysis

    %global variables
  global OptShape
  global ResearchLoadFACTOR

  TempLoadFACTOR = 1;
  LambdaMin = 1; %value that verifies the arch (safe arch)
  LambdaMax = 100000; %does not verify the arch (unsafe arch)
  if OptShape.Type_Analysis == 1 %1 = 'FullRangeFactor' = our modified method (FRS)
      while (LambdaMax - LambdaMin) >= 0.000000001 | OptShape.sID <= 0
          if ResearchLoadFACTOR == 1  %YES

             TempLoadFACTOR = (LambdaMin + LambdaMax) / 2;
          end
            %computes the line of thrust closet to the geometrical axis
          BestThrustLine(TempLoadFACTOR)
            %computes the Full Range Factor of Safety (our modified method - FRS)
          FullRangeFactor
          if TempLoadFACTOR == 1
              break
          end
          if ResearchLoadFACTOR == 1  %YES
              if OptShape.sID > 0 %safe arch
                  LambdaMin = TempLoadFACTOR;
              else %unsafe arch
                    LambdaMax = TempLoadFACTOR;
              end
          end
      end
  else % 2 = 'GeometricalFactorOfSafety' = GFS method by Heyman
      while (LambdaMax - LambdaMin) >= 0.000000001 | OptShape.sID >= OptShape.RealArchMinimumThickness
          if ResearchLoadFACTOR == 1  %YES
              TempLoadFACTOR = (LambdaMin + LambdaMax) / 2;
          end
            %computes the line of thrust closet to the geometrical axis
          BestThrustLine(TempLoadFACTOR)
            %computes the Geometrical Factor of Safety (GFS by Heyman)
          GeomFactor
          if TempLoadFACTOR == 1
              break
          end
          if ResearchLoadFACTOR == 1  %YES
              if OptShape.sID < OptShape.RealArchMinimumThickness %safe arch
                  LambdaMin = TempLoadFACTOR;
              else %unsafe arch
                  LambdaMax = TempLoadFACTOR;
              end
          end
      end
  end
  OptShape.LoadFactor = TempLoadFACTOR;
return
```

The “BestThrustLine” routine

This routine computes the line of thrust closest to the geometrical axis and must be saved to the ‘BestThrustLine.m’ file. At the beginning the global variables used herein are declared.

```
function BestThrustLine(TempLoadFACTOR)

  global MaxNumBrick
  global Brick
  global VectorF
  global OptShape
```

Then, the ComputeLoadVectorF routine, to which the value of the load factor is passed by the TempLoadFACTOR variable, is called.

```
%compute the load vector {F}
ComputeLoadVectorF(TempLoadFACTOR)
```

The lines that follow implement the mathematical formulation of the procedure, that builds and solves the system of linear equations that provides the coordinates of the vertices of the line of thrust. The theoretical background of the method is briefly reported in the second section, but a detailed reference of the mathematical procedure is found in [41].

```
  %*************************************************************************************
  % Search for the line of thrust closest to the geometrical axis of the arch,
  % assuming the poly-line through the element centroids to be the axis of the arch
  %*************************************************************************************

  %Compute vector H collected from the horizontal distances hi between the action lines of the load vectors.
  %Point A (h1) is the rightmost point of the arch and point B (h_last)the leftmost one.
TempAbutmentDX=[Brick(1).x(1), Brick(1).x(2)];
Vector_H(1) = max(TempAbutmentDX) - Brick(1).xG;
for i = 2 : MaxNumBrick
    Vector_H(i) = Brick(i - 1).xG - Brick(i).xG;
end
TempAbutmentSX=[Brick(MaxNumBrick).x(3), Brick(MaxNumBrick).x(4)];
Vector_H(MaxNumBrick + 1) = Brick(MaxNumBrick).xG - min(TempAbutmentSX);

   %build the general matrix [D], symmetric, with entries collected from vector {H}
for indexRow = 1 : MaxNumBrick
    for indexColumn = indexRow : indexRow + 1
        if indexRow == indexColumn
            %entries of the main diagonal
            Matrix_D(indexRow, indexColumn) = (Vector_H(indexRow) + Vector_H(indexRow + 1)) / (Vector_H(indexRow) * Vector_H(indexRow +
1));
        else
            if indexColumn > MaxNumBrick
                break
            end
            %entries in the upper triangle
            Matrix_D(indexRow, indexColumn) = -1 / Vector_H(indexColumn);
            %entries in the lower triangle
            Matrix_D(indexColumn, indexRow) = Matrix_D(indexRow, indexColumn);
        end
    end
end

   %build the vertical load vector {T1}
   %(it obtains the values from the load vector {F})
for indexRow = 1 : MaxNumBrick
    Vector_T1(indexRow, 1) = VectorF(indexRow * 3 - 1);
end

   %build vector {T2}, whose only nonzero entry is 1/h1 (i.e. the first entry)
Vector_T2 = zeros(MaxNumBrick,1);
Vector_T2(1, 1) = 1 / Vector_H(1);

   %build vector {T3}, whose only nonzero entry is 1/h_last (i.e. the last entry)
Vector_T3 = zeros(MaxNumBrick,1);
Vector_T3(MaxNumBrick, 1) = 1 / Vector_H(length(Vector_H));

   %compute the inverse of matrix [D]: [Dinv]
Matrix_Dinv = inv(Matrix_D);

   %                              -1
   %compute the product {R1} = [D] * {T1}
Vector_R1 = Matrix_Dinv * Vector_T1;

   %                              -1
   %compute the product {R2} = [D] * {T2}
Vector_R2 = Matrix_Dinv * Vector_T2;

   %                              -1
   %compute the product {R3} = [D] * {T3}
Vector_R3 = Matrix_Dinv * Vector_T3;

   %build matrix [N], symmetric, with entries collected from vectors {R1},{R2},{R3}
Matrix_N = zeros(3,3);
SumR1 = 0; SumR2 = 0; SumR3 = 0;
SumR1R2 = 0; SumR1R3 = 0; SumR2R3 = 0;
for i = 1 : length(Vector_R1)
    SumR1 = SumR1 + Vector_R1(i, 1) ^ 2;
    SumR2 = SumR2 + Vector_R2(i, 1) ^ 2;
    SumR3 = SumR3 + Vector_R3(i, 1) ^ 2;
    SumR1R2 = SumR1R2 + Vector_R1(i, 1) * Vector_R2(i, 1);
    SumR1R3 = SumR1R3 + Vector_R1(i, 1) * Vector_R3(i, 1);
    SumR2R3 = SumR2R3 + Vector_R2(i, 1) * Vector_R3(i, 1);
end
Matrix_N(1, 1) = SumR1;
Matrix_N(2, 2) = SumR2;
Matrix_N(3, 3) = SumR3;
Matrix_N(1, 2) = SumR1R2;
Matrix_N(1, 3) = SumR1R3;
Matrix_N(2, 3) = SumR2R3;
Matrix_N(2, 1) = Matrix_N(1, 2);
Matrix_N(3, 1) = Matrix_N(1, 3);
Matrix_N(3, 2) = Matrix_N(2, 3);

   %build vector {W} and collect it with the least squares entries
SumR1G = 0; SumR2G = 0; SumR3G = 0;
for i = 1 : MaxNumBrick
    SumR1G = SumR1G + Vector_R1(i, 1) * Brick(i).yG;
    SumR2G = SumR2G + Vector_R2(i, 1) * Brick(i).yG;
    SumR3G = SumR3G + Vector_R3(i, 1) * Brick(i).yG;
end
Vector_W(1, 1) = SumR1G;
Vector_W(2, 1) = SumR2G;
Vector_W(3, 1) = SumR3G;

   %compute the inverse of matrix [N]: [Ninv]
Matrix_Ninv = inv(Matrix_N);

   %                               -1
   %compute the product {P} =  [N] * {W}
Vector_P = Matrix_Ninv * Vector_W;

   %solve the matrix system {Y} = {R1}K + {R2}yA + {R3}yB
   %and provide entries of vector {Y}, that is the heights (i.e. ordinates) of vertices of the line of thrust
OptShape. Vector_Y = zeros(length(Vector_R1),1);
for i = 1 : length(OptShape.Vector_Y)
    OptShape.Vector_Y(i, 1) = Vector_R1(i, 1) * Vector_P(1, 1) + Vector_R2(i, 1) * Vector_P(2, 1) + Vector_R3(i, 1) * Vector_P(3, 1);
end
   %input the first and last vertices of the line of thrust (points A and B) in vector {Y}
TempVector_Y = OptShape.Vector_Y;
OptShape.Vector_Y=[Vector_P(2, 1); TempVector_Y; Vector_P(3, 1)];
   %store the value of the thrust H
OptShape.ThrustH = 1 / Vector_P(1, 1);

   %compute vector {X}, that collects the abscissae of vertices of the line of thrust
OptShape.Vector_X=[max(TempAbutmentDX), Brick.xG, min(TempAbutmentSX)];
OptShape.Vector_X=OptShape.Vector_X';
return
```

The “ComputeLoadVectorF” routine

This routine computes the load vector {F} of the structure under analysis. It must be saved to the ‘ComputeLoadVectorF.m’ file by the user. The dimension of the load vector is 3 times the number of elements of the arch, because for each element the horizontal force, the vertical force and the moment as respects to the centroid must be defined. Nevertheless, the current release of the method considers only vertical loads, applied to the element centroids. Therefore, the horizontal force and the moment are zero, while the vertical force is given by the weight of the element plus the value of an additional vertical force, that the user can input in the model when he wants to compute also the load factor.

```
function ComputeLoadVectorF(TempSafetyFACTOR)

   %global variables
  global Brick
  global MaxNumBrick
  global VectorF

  for indexBrick = 1 : MaxNumBrick
        %load vector {F}
         %Horizontal forces Fx (Kgf): equal to zero
      VectorF(indexBrick * 3 - 2) = 0;
          %Vertical forces Fy (Kgf): weight of the element + additional vertical forces inputted by the user
      VectorF(indexBrick * 3 - 1) = -(Brick(indexBrick).Weight - Brick(indexBrick).Fy * TempSafetyFACTOR) * sin(pi / 2 + pi);
          %Moments (Kgf*cm): equal to zero
      VectorF(indexBrick * 3) = 0;
  end
return
```

The “FullRangeFactor” routine

Two routines must be saved to the ‘FullRangeFactor.m’ file. The “FullRangeFactor” routine is the main function, that computes the lower and upper bound of the domain of equilibrium thrust lines and the performance factor. The ‘IntersectionVerticalLine_GenericLine’ function is used to compute the intersection point of two straight lines.

Let us describe the main routine. First the global variables are declared. Then the code calls the ‘IntersectionVerticalLine_GenericLine’ function that computes the intersection points of the left (right) interface and the line of thrust and, as a consequence, returns the points of pressure in the left (right) interface.

```
function FullRangeFactor
    %global variables
  global Brick
  global MaxNumBrick
  global OptShape

    %compute the coordinates of the points of pressure in the right and left interfaces of each element
  for indexBrick = 1 : MaxNumBrick
        %right interface of the element
      alfa = atan((Brick(indexBrick).y(2) - Brick(indexBrick).y(1)) / (Brick(indexBrick).x(2) - Brick(indexBrick).x(1)));
      [OptShape.Vector_xCPdx(indexBrick),  OptShape.Vector_yCPdx(indexBrick)]  =  LinesIntersection  (OptShape.Vector_X(indexBrick,  1),
OptShape.Vector_Y(indexBrick,  1),  OptShape.Vector_X(indexBrick+1,  1),  OptShape.Vector_Y(indexBrick+1,  1),  Brick(indexBrick).x(1),
Brick(indexBrick).y(1), alfa);
        %left interface of the element

      alfa = atan((Brick(indexBrick).y(3) - Brick(indexBrick).y(4)) / (Brick(indexBrick).x(3) - Brick(indexBrick).x(4)));
      [OptShape.Vector_xCPsx(indexBrick),  OptShape.Vector_yCPsx(indexBrick)]  =  LinesIntersection  (OptShape.Vector_X(indexBrick+1,  1),
OptShape.Vector_Y(indexBrick+1, 1), OptShape.Vector_X(indexBrick + 2, 1), OptShape.Vector_Y(indexBrick + 2, 1), Brick(indexBrick).x(4),
Brick(indexBrick).y(4), alfa);
  end
```

Then, in the following code the lower bound of the domain is computed. The lower bound is defined as that line of thrust obtained by shifting vertically the line of thrust closest to the geometrical axis until it becomes tangent to the intrados curve of the arch. In the following lines of the code, the step by step algorithm for computing the lower bound line of thrust, developed by the authors, is reported and described.

```
%*********************************************************************
%*       Search for the LIMITING LOWER LINE OF THRUST: PF-a          *
%* It is the line of thrust that is all above the intrados poly-line *
%*********************************************************************
```

The code that follows searches for the segments of the thrust line that are intercepted by the vertical lines (hereafter referred to as scanning lines) passing through the intrados point of the right interface of each element (point number 1). For the generic scanning line *i*, the segment of the thrust

line intercepted by that line is detected by comparing the abscissa of point 1, through which the scanning line passes, and the abscissa of the two end points *i* and *i+1* of each segment of the thrust line. If abscissa of point 1 (referred to as x1 in the code) is comprised between abscissa of point *i+1* and abscissa of point *i*, then the algorithm has found the segment of the thrust line intercepted by the scanning line and the 'IntersectionVerticalLine_GenericLine' function is called. This function returns the coordinates of the intersection point of the segment detected and the scanning line, and they are stored in the variables [xK,yK]. Finally, the vertical distance vector between the intersection point K and point 1 are computed and stored in the vector Vector_dINF. This check is repeated testing all segments of the thrust line (for i = 1: length(OptShape.Vector_Y)-1) on the i-th scanning line and the loop is performed for all the scanning lines.

```
  for indexBrick = 1 : MaxNumBrick
      %search for the segment of the thrust line that is intercepted by the vertical line
      %passing through the intrados point of the right interface of the element
    for i = 1 : length(OptShape.Vector_Y)-1 %number of segments of the thrust line
        IntersecFound=0;
        if Brick(indexBrick).x(1) <= OptShape.Vector_X(i, 1) & Brick(indexBrick).x(1) >= OptShape.Vector_X(i+1, 1)
             %you have found the segment of the thrust line intercepted by the vertical line:
             %compute the intersection point of this segment and the vertical line passing
             %through the intrados vertex in the right interface of the element
          [xK,yK] = IntersectionVerticalLine_GenericLine (OptShape.Vector_X(i, 1), OptShape.Vector_Y(i, 1), OptShape.Vector_X(i+1, 1),
OptShape.Vector_Y(i+1, 1), Brick(indexBrick).x(1));
          IntersecFound=1;
            %now compute the vertical distance (may be positive or negative because it's a vector)
            %between the intersection point and the intrados point
          Vector_dINF(indexBrick * 2 - 1) = Brick(indexBrick).y(1) - yK;
          break
        end
    end
      %if you have not found an intersection point, build an ideal point at infinity (negative)
    if IntersecFound==0
        Vector_dINF(indexBrick * 2 - 1) = 1E-24; %realmin
    end
```

The procedure above is repeated for detecting also the segments of the thrust line that are intercepted by the vertical lines passing through the intrados point of the left interface of each element (point number 4).

```
      %search for the segment of the thrust line that is intercepted by the vertical line
      %passing through the intrados point of the left interface of the element
    for i = 1 : length(OptShape.Vector_Y)-1 %number of segments of the thrust line
        IntersecFound=0;
        if Brick(indexBrick).x(4) <= OptShape.Vector_X(i, 1) & Brick(indexBrick).x(4) >= OptShape.Vector_X(i+1, 1)
             %you have found the segment of the thrust line intercepted by the vertical line:
             %compute the intersection point of this segment and the vertical line passing
             %through the intrados vertex in the left interface of the element
          [xK,yK] = IntersectionVerticalLine_GenericLine (OptShape.Vector_X(i, 1), OptShape.Vector_Y(i, 1), OptShape.Vector_X(i+1, 1),
OptShape.Vector_Y(i+1, 1), Brick(indexBrick).x(4));
          IntersecFound=1;
            %now compute the vertical distance (may be positive or negative because it's a vector)
            %between the intersection point and the intrados point
          Vector_dINF(indexBrick * 2) = Brick(indexBrick).y(4) - yK;
          break
        end
    end
      %if you have not found an intersection point, build an ideal point at infinity (negative)
    if IntersecFound==0
        Vector_dINF(indexBrick * 2) = 1E-24; %realmin
    end
  end
```

Finally, lines that follow search for the minimum distance vector, among all the distances that have been stored in the vector Vector_dINF.

```
Vector_dINF.

    %Compute the shortest distance vector, vertically arranged from
    %point K of the thrust line to the intrados point of the arch
  [dINFmax,j]  =  max(Vector_dINF); %it's the minimum distance
  OptShape.sINF = dINFmax;
```

In the following lines, the minimum distance is used to compute the ordinates of the lower bound thrust line that are stored in the global vector OptShape.Vector_Yinf.

```
    %Compute the coordinates of the limit thrust line tangent to the intrados: PF-a
  for i = 1 : length(OptShape.Vector_Y)
    OptShape.Vector_Yinf(i, 1) = OptShape.Vector_Y(i, 1) + OptShape.sINF;
  end
```

Then, in the following code the upper bound of the domain is computed. Coherently with the lower bound defined above, the upper bound is defined as that line of thrust obtained by shifting vertically

the line of thrust closest to the geometrical axis until it becomes tangent to the extrados curve of the arch. Therefore, the scanning lines are defined to pass through the extrados points of the arch (points 2 and 3). The following lines that compute the upper bound line of thrust are identical to the code described above and, therefore, it is not necessary to comment on them.

```
%**********************************************************************
%*       Search for the LIMITING UPPER LINE OF THRUST: PF-b           *
%* It is the line of thrust that is all under the extrados poly-line *
%**********************************************************************

 for indexBrick = 1 : MaxNumBrick
      %search for the segment of the thrust line that is intercepted by the vertical line
      %passing through the extrados point of the right interface of the element
    for i = 1 : length(OptShape.Vector_Y)-1 %number of segments of the thrust line
        IntersecFound=0;
        if Brick(indexBrick).x(2) <= OptShape.Vector_X(i, 1) & Brick(indexBrick).x(2) >= OptShape.Vector_X(i+1, 1)
            %you have found the segment of the thrust line intercepted by the vertical line:
            %compute the intersection point of this segment and the vertical line passing
            %through the extrados vertex in the right interface of the element
          [xK,yK] = IntersectionVerticalLine_GenericLine (OptShape.Vector_X(i, 1), OptShape.Vector_Y(i, 1), OptShape.Vector_X(i+1, 1),
OptShape.Vector_Y(i+1, 1), Brick(indexBrick).x(2));
            IntersecFound=1;
              %now compute the vertical distance (may be positive or negative because it's a vector)
              %between the intersection point and the extrados point
            Vector_dSUP(indexBrick * 2 - 1) = Brick(indexBrick).y(2) - yK;
            break
        end
    end
       %if you have not found an intersection point, build an ideal point at infinity (positive)
    if IntersecFound==0
        Vector_dSUP(indexBrick * 2 - 1) = 1E+24; %realmax
    end

      %search for the segment of the thrust line that is intercepted by the vertical line
      %passing through the extrados point of the left interface of the element
    for i = 1 : length(OptShape. Vector_Y)-1 %number of segments of the thrust line
        IntersecFound=0;
        if Brick(indexBrick).x(3) <= OptShape.Vector_X(i, 1) & Brick(indexBrick).x(3) >= OptShape.Vector_X(i+1, 1)
            %you have found the segment of the thrust line intercepted by the vertical line:
            %compute the intersection point of this segment and the vertical line passing
            %through the extrados vertex in the left interface of the element
          [xK, yK] = IntersectionVerticalLine_GenericLine (OptShape.Vector_X(i, 1), OptShape.Vector_Y(i, 1), OptShape.Vector_X(i+1, 1),
OptShape.Vector_Y(i+1, 1), Brick(indexBrick).x(3));
            IntersecFound=1;
              %now compute the vertical distance (may be positive or negative because it's a vector)
              %between the intersection point and the extrados point
            Vector_dSUP(indexBrick * 2) = Brick(indexBrick).y(3) - yK;
            break
        end
    end
       %if you have not found an intersection point, build an ideal point at infinity (positive)
    if IntersecFound==0
      Vector_dSUP(indexBrick * 2) = 1E+24; %realmax
    end
 end

   %Compute the shortest distance vector, vertically arranged from
   %point K of the thrust line to the extrados point of the arch
 [dSUPmin,i]  =  min(Vector_dSUP);
 OptShape.sSUP = dSUPmin;

   %Compute the coordinates of the limit thrust line tangent to the extrados: PF-b

 for i = 1 : length(OptShape.Vector_Y)
     OptShape.Vector_Ysup(i, 1) = OptShape.Vector_Y(i, 1) + OptShape.sSUP;
 end
```

Line that follows computes the minimum thickness of the domain (vertical distance between the lower and upper bound thrust lines) and stores it to the OptShape.sID variable. If the thickness is positive, the domain exists and the arch is safe; if it is negative the domain does not exist and the arch is unsafe.

```
  %Compute the thickness (that is constant because vertically measured)
  %of the domain within the real one. Vector Ysup(1)-Yinf(1) is defined
  %in such a way that, if it is negative, no thrust line of that shape can lie
  %within the profile of the arch (i.e. the domain is negative)
OptShape.sID = OptShape.Vector_Ysup(1, 1) - OptShape.Vector_Yinf(1, 1);
```

The last part of the code computes the safety factor. The minimum vertical thickness in correspondence to the action lines of the load vectors is computed and stored in the variable OptShape.RealArchMinimumThickness. Then, the full range factor of safety is computed and stored in the OptShape.Geom_Safety_Factor variable. The performance factor, that is the reciprocal of the full range factor, is not computed in this routine but it is calculated directly in the main routine and is showed in both the graphical output results and the legend.

```
    %Create the close contour of the profile of the arch: build the two vectors collected
    %from the coordinates (x,y) of points that define the contour
  i = 0;
  for indexBrick = 1 : MaxNumBrick
    i = i + 1;
    if indexBrick == 1
      Contour_X(i) = Brick(indexBrick).x(1);
      Contour_Y(i) = Brick(indexBrick).y(1);
      i = i + 1;
      Contour_X(i) = Brick(indexBrick).x(2);
      Contour_Y(i) = Brick(indexBrick).y(2);
       i = i + 1;
      Contour_X(i) = Brick(indexBrick).x(3);
      Contour_Y(i) = Brick(indexBrick).y(3);
    else
      Contour_X(i) = Brick(indexBrick).x(3);
      Contour_Y(i) = Brick(indexBrick).y(3);
    end
  end
  for indexBrick = MaxNumBrick : -1 : 1
    if indexBrick == 1
      i = i + 1;
      Contour_X(i) = Brick(indexBrick).x(4);
      Contour_Y(i) = Brick(indexBrick).y(4);
      i = i + 1;
      Contour_X(i) = Brick(indexBrick).x(1);
      Contour_Y(i) = Brick(indexBrick).y(1);
    else
      i = i + 1;
      Contour_X(i) = Brick(indexBrick).x(4);
      Contour_Y(i) = Brick(indexBrick).y(4);
    end
  end

    %Search for the two points of intersection between each vertical line passing
    %through the element centroid and the contour (i.e. the profile) of the arch
  for indexBrick = 1 : MaxNumBrick
      %Search for the segment of the poly-line defining the arch contour that is intercepted by the
      %vertical line passing through the element centroid (i.e. the action line of the load)
    PointFound = 0;
    for i = 1 : length(Contour_X)-1 %it's the number of segments defining the arch contour
      TempX=[ Contour_X(i), Contour_X(i + 1)];
      minX = min(TempX);
      maxX = max(TempX);
      if Brick(indexBrick).xG <= maxX & Brick(indexBrick).xG > minX
          %you have found the segment of the arch contour intercepted by the vertical line:
          %compute the intersection point of this segment and the vertical line passing
          %through the element centroid
        PointFound = PointFound + 1;
        [xK,   yK]   =   IntersectionVerticalLine_GenericLine   (Contour_X(i),   Contour_Y(i),   Contour_X(i   +   1),   Contour_Y(i   +   1),
Brick(indexBrick).xG);
       if PointFound == 1
         xZ = xK;
         yZ = yK;
       end
      end
    end
      %Compute the vertical distance between the two points, that is the
      %vertical thickness in correspondence to the straight line y=xG(indexBrick)
    vector_sVERT(indexBrick) = abs(yZ - yK);

   end

     %*********************************************************************
     %*         Compute the GEOMETRICAL FACTOR OF SAFETY                  *
     %*       (herein renamed FULL RANGE FACTOR OF SAFETY)                *
     %* It is the ratio between the minimum arch thickness (vertically *
     %* measured in correspondence to the action lines of the loads)    *
     %* and the thickness of the domain (vertically measured            *
     %* as the distance between the two limit thrust lines)             *
     %*********************************************************************

     %Search for the minimum thickness among all vertical thicknesses
   [minValue,K] = min(Vector_sVERT);
   OptShape.RealArchMinimumThickness = Vector_sVERT(K);
   OptShape.Geom_Safety_Factor = Vector_sVERT(K) / OptShape.sID;

 return

 function [xP,yP] = IntersectionVerticalLine_GenericLine(x1, y1, x2, y2, K)
     %compute the intersection point of two straight lines:
     %straight line r: from two points (1-2)
     %straight line s:          line y = k

     % / y = k                    (s)
     % |
     %<  y -y1   x- x1        --> P=(xP|yP)
     % | ----- = -----             (r)
     % \ y2-y1   x2-x1

   if (x2 - x1) ~= 0
       %generic straight line r (inclined or horizontal, but not vertical)
     xP = K;
     yP = ((y2 - y1) * K + x1 * (y1 - y2) + y1 * (x2 - x1)) / (x2 - x1);
   else
       %vertical straight line r, therefore parallel to line s:
   end
  return
```

The “GeomFactor” routine

This routine, that must be saved to the ‘GeomFactor.m’ file, computes the safety factor according to the original Heymanian theory based on the research of the arch of minimal thickness within the profile of the actual arch capable of supporting the same system of forces.

As for the “FullRangeFactor” routine above, after the global variables used herein are declared, the coordinates of the points of pressure in the right and left interfaces are computed.

```
function GeomFactor
    %global variables
  global Brick
  global MaxNumBrick
  global OptShape

    %compute the coordinates of the points of pressure on the right and left interfaces of each element
  for indexBrick = 1 : MaxNumBrick
       %right interface of the element
      alfa = atan((Brick(indexBrick).y(2) - Brick(indexBrick).y(1)) / (Brick(indexBrick).x(2) - Brick(indexBrick).x(1)));
      [OptShape.Vector_xCPdx(indexBrick),  OptShape.Vector_yCPdx(indexBrick)]  =  LinesIntersection  (OptShape.Vector_X(indexBrick,  1),
OptShape.Vector_Y(indexBrick,  1),  OptShape.Vector_X(indexBrick+1,  1),  OptShape.Vector_Y(indexBrick+1,  1),  Brick(indexBrick).x(1),
Brick(indexBrick).y(1), alfa);
       %left interface of the element
      alfa = atan((Brick(indexBrick).y(3) - Brick(indexBrick).y(4)) / (Brick(indexBrick).x(3) - Brick(indexBrick).x(4)));
      [OptShape.Vector_xCPsx(indexBrick),  OptShape.Vector_yCPsx(indexBrick)]  =  LinesIntersection  (OptShape.Vector_X(indexBrick+1,  1),
OptShape.Vector_Y(indexBrick+1, 1), OptShape.Vector_X(indexBrick + 2, 1), OptShape.Vector_Y(indexBrick + 2, 1), Brick(indexBrick).x(4),
Brick(indexBrick).y(4), alfa);
  end
```

In the following lines, the thickness of the arch of minimal thickness within the real one is searched. To perform this search, two vectors are defined for each joint (i.e. for both the left and right interface of each element): vector oriented from the centroid of the joint to the point of pressure (referred to as Vector_dSUP in the code) and vector oriented from the point of pressure to the extrados (referred to as Vector_sSUP in the code) in the case in which the line of thrust is above the geometrical axis; vector oriented from the centroid of the joint to the point of pressure (referred to as Vector_dINF in the code) and vector oriented from the point of pressure to the intrados (referred to as Vector_sINF in the code) in the case in which the line of thrust is under the geometrical axis. The lines that follow perform this computation.

```
 %compute the thickness of the arch of minimal thickness within the real one

 Vector_dSUP=[]; Vector_sSUP=[];
 Vector_dINF=[]; Vector_sINF=[];
 for indexBrick = 1 : MaxNumBrick

     %If the point of pressure is below the geometrical axis, compute the vector
     %collecting all the superior distances, that is the distances between
     %the points of pressure and the extrados points of the joints.
     %But, if the point of pressure is above the geometrical axis, compute the
     %vector collecting all the inferior distances, that is the distances between
     %the points of pressure and the intrados points of the joints.
     %right interface of the element
    xGdx = (Brick(indexBrick).x(1) + Brick(indexBrick).x(2)) / 2;
    yGdx = (Brick(indexBrick).y(1) + Brick(indexBrick).y(2)) / 2;
    if sign(Brick(indexBrick).x(2) - xGdx) == sign(OptShape.Vector_xCPdx(indexBrick) - xGdx) & sign(Brick(indexBrick).y(2) - yGdx) ==
sign(OptShape.Vector_yCPdx(indexBrick) - yGdx)
       %if vector Gi-->2 has the same sense of direction of vector Gi-->CPdx, then the CP is above the geometrical axis: compute the
distance dSUP
      indexRow = length(Vector_dSUP)+1;
      Vector_dSUP(indexRow)  =  sqrt((Brick(indexBrick).x(2)  -  OptShape.Vector_xCPdx(indexBrick))  ^  2  +  (Brick(indexBrick).y(2)  -
OptShape.Vector_yCPdx(indexBrick)) ^ 2);
      Vector_sSUP(indexRow) = sqrt((OptShape.Vector_xCPdx(indexBrick) - (Brick(indexBrick).x(1) + Brick(indexBrick).x(2)) / 2) ^ 2 +
(OptShape.Vector_yCPdx(indexBrick) - (Brick(indexBrick).y(1) + Brick(indexBrick).y(2)) / 2) ^ 2);
    else
       %else, the point of pressure is below the geometrical axis: compute the distance dINF
      indexRow = length(Vector_dINF)+1;
      Vector_dINF(indexRow)  =  sqrt((Brick(indexBrick).x(1)  -  OptShape.Vector_xCPdx(indexBrick))  ^  2  +  (Brick(indexBrick).y(1)  -
OptShape.Vector_yCPdx(indexBrick)) ^ 2);
      Vector_sINF(indexRow) = sqrt((OptShape.Vector_xCPdx(indexBrick) - (Brick(indexBrick).x(1) + Brick(indexBrick).x(2)) / 2) ^ 2 +
(OptShape.Vector_yCPdx(indexBrick) - (Brick(indexBrick).y(1) + Brick(indexBrick).y(2)) / 2) ^ 2);
    end

     %left interface of the element
    xGsx = (Brick(indexBrick).x(3) + Brick(indexBrick).x(4)) / 2;
    yGsx = (Brick(indexBrick).y(3) + Brick(indexBrick).y(4)) / 2;
    if sign(Brick(indexBrick).x(3) - xGsx) == sign(OptShape.Vector_xCPsx(indexBrick) - xGsx) & sign(Brick(indexBrick).y(3) - yGsx) ==
sign(OptShape.Vector_yCPsx(indexBrick) - yGsx)
       %if vector Gi-->3 has the same sense of direction of vector Gi-->CPsx, then the CP is above the geometrical axis: compute the
distance dSUP
      indexRow = length(Vector_dSUP)+1;
      Vector_dSUP(indexRow)  =  sqrt((Brick(indexBrick).x(3)  -  OptShape.Vector_xCPsx(indexBrick))  ^  2  +  (Brick(indexBrick).y(3)  -
OptShape.Vector_yCPsx(indexBrick)) ^ 2);
      Vector_sSUP(indexRow) = sqrt((OptShape.Vector_xCPsx(indexBrick) - (Brick(indexBrick).x(3) + Brick(indexBrick).x(4)) / 2) ^ 2 +
(OptShape.Vector_yCPsx(indexBrick) - (Brick(indexBrick).y(3) + Brick(indexBrick).y(4)) / 2) ^ 2);
    else
       %else, the point of pressure is below the geometrical axis: compute the distance dINF
      indexRow = length(Vector_dINF)+1;
      Vector_dINF(indexRow)  =  sqrt((Brick(indexBrick).x(4)  -  OptShape.Vector_xCPsx(indexBrick))  ^  2  +  (Brick(indexBrick).y(4)  -
OptShape.Vector_yCPsx(indexBrick)) ^ 2);
      Vector_sINF(indexRow) = sqrt((OptShape.Vector_xCPsx(indexBrick) - (Brick(indexBrick).x(3) + Brick(indexBrick).x(4)) / 2) ^ 2 +
(OptShape.Vector_yCPsx(indexBrick) - (Brick(indexBrick).y(3) + Brick(indexBrick).y(4)) / 2) ^ 2);
    end
 end
```

The lines that follow are to compute the longest superior thickness vector (maximum value of Vector_sSUP stored in the variable sSUPmax) and the longest inferior thickness vector (minimum value of Vector_sINF stored in the variable sINFmax) and they are assigned to the variables OptShape. sSUP and OptShape.sINF respectively. They are also used to compute the thickness of the ideal arch (OptShape.sID).

```
  %Compute the maximum sINF and the minimum sSUP
  %(but if the l.o.t. is entirely above the geometrical axis, then sSUP=0 and the vector
  %of superior vertices is empty; conversely, if the l.o.t. is entirely under the
  %geometrical axis, then sINF=0 and the vector of inferior vertices is empty)
if length(Vector_sSUP) > 0
  [sSUPmax,i]  =  max(Vector_sSUP);
  OptShape.sSUP = Vector_sSUP(i);
else
  OptShape.sSUP =0;
end
if length(Vector_sINF) > 0
  [sINFmax,j]  =  max(Vector_sINF);
  OptShape.sINF = Vector_sINF(j);
else
  OptShape.sINF =0;
end

  %compute the (constant) thickness of the arch of minimal thickness
  %within the real arch as the sum of sSUP + sINF
OptShape.sID = OptShape.sSUP + OptShape.sINF;
```

The last lines of the code are devoted to the computation of the geometrical factor of safety, obtained by the ratio between the minimum thickness of all joints and the thickness of the arch of minimal thickness within the actual one.

```
    %compute the geometrical factor of safety, given by the ratio between
    %the thickness of the arch (that is the minimum thickness among all
    %joint thickness in the case of a variable thickness arch) and the
    %thickness of the arch of minimal thickness within the actual one
  K = 0;
  for indexBrick = 1 : MaxNumBrick
     %compute the minimum thickness among all joint thickness
    K = K + 1;
    Vector_JointThickness(K)   =   sqrt((Brick(indexBrick).x(2)   -   Brick(indexBrick).x(1))   ^   2   +   (Brick(indexBrick).y(2)   -
Brick(indexBrick).y(1)) ^ 2);
    K = K + 1;
    Vector_JointThickness(K)   =   sqrt((Brick(indexBrick).x(3)   -   Brick(indexBrick).x(4))   ^   2   +   (Brick(indexBrick).y(3)   -
Brick(indexBrick).y(4)) ^ 2);
  end
  [minValue,K]  =  min(Vector_JointThickness);
  OptShape.RealArchMinimumThickness = Vector_JointThickness(K);
    %geometrical factor of safety
  OptShape.Geom_Safety_Factor = Vector_JointThickness(K) / OptShape.sID;
return
```

The “LinesIntersection” and “OrientedLineAngle” functions

These two service functions complete the Matlab code and must be saved to the ‘LinesIntersection.m’and ‘OrientedLineAngle.m’ files respectively. The “LinesIntersection” function is called both by the “FullRangeFactor” and “GeomFactor” routines to compute the coordinates of the points of pressure in the left and right interface of each joint. The coordinates of two points (x1,y1;x2,y2) belonging to the first straight line and a point (xO,yO) plus the inclination angle (alfa) of the second straight line are passed to this function. The function returns the coordinates of the intersection point ([xP,yP] in the code).

```
function [xP,yP] = LinesIntersection(x1, y1, x2, y2, xO, yO, alfa)
    %compute the intersection point of two lines:
    %line r: defined passing through two points (1-2)
    %line s: defined for one point (O) and by its
    %gradient (m)
    % / y -yO = m(x - xO)       (s)
    % |
    %<  y -y1   x- x1        --> P=(xP|yP)
    % | ----- = -----           (r)
    % \ y2-y1   x2-x1

  m = tan(alfa);
  xP = (m * xO * (x2 - x1) - yO * (x2 - x1) + y1 * (x2 - x1) - x1 * (y2 - y1)) / (m * (x2 - x1) - (y2 - y1));
  if alfa ~= pi / 2  & alfa ~= 3 * pi / 2
        %line s: horizontal or inclined
        %line r: any
      yP = m * (xP - xO) + yO;
  else
        %line s: vertical
      if (x2 - x1) ~= 0
         %line r: any
        yP = (y1 * (x2 - x1) + xP * (y2 - y1) - x1 * (y2 - y1)) / (x2 - x1);
      else
          %line r: horizontal
        yP = y1;
      end
  end
return
```

The “OrientedLineAngle” function is used by the main section of the program to plot the Heymanian arch of minimal thickness. The horizontal and vertical components of the direction vector of a straight line (DeltaX, DeltaY) are passed to the function that returns the inclination angle of that line in the interval [0-2Π].

```
function [LineAngle] = OrientedLineAngle(DeltaX, DeltaY)
    %*********************************************
    %* returns the positive angle of an oriented  *
    %* straight line in the interval 0 --- 2PI.   *
    %*********************************************

    %DeltaX and DeltaY are: the difference between the coordinates
    %of the end point and the start point of the direction vector
    %of a straight line.
  if DeltaY > 0
      if DeltaX > 0
        LineAngle = atan(DeltaY / DeltaX);
      elseif DeltaX < 0
        LineAngle = atan(DeltaY / DeltaX);
        LineAngle = pi - (-LineAngle);
       elseif DeltaX == 0
         LineAngle = pi / 2;
       end
   elseif DeltaY < 0
       if DeltaX > 0
         LineAngle = atan(DeltaY / DeltaX);
         LineAngle = 2 * pi - (-LineAngle);
       elseif DeltaX < 0
         LineAngle = atan(DeltaY / DeltaX);
         LineAngle = pi + LineAngle;
       elseif DeltaX == 0
         LineAngle = 3 * pi / 2;
       end
   elseif DeltaY == 0
       if DeltaX > 0
         LineAngle = 0;
       elseif DeltaX < 0
         LineAngle = pi;
       end
   end
 return
```

Appendix A. Supplementary data

Supplementary material related to this article can be found, in the online version, at doi:https://doi.org/10.1016/j.mex.2019.05.033.

References

[1] J. Heyman, The safety of masonry arches, Int. J. Mech. Sci. 11 (4) (1969) 363–385, doi:http://dx.doi.org/10.1016/0020-7403(69)90070-8.

[2] S. Galassi, N. Ruggieri, G. Tempesta, A novel numerical tool for seismic vulnerability analysis of ruins in archaeological sites, Int. J. Archit. Herit. (2018), doi:http://dx.doi.org/10.1080/15583058.2018.1492647.

[3] (a) S. Galassi, N. Ruggieri, G. Tempesta, Ruins and archaeological artifacts: vulnerabilities analysis for their conservation through the original computer program BrickWORK, Structural Analysis of Historical Constructions - Proceedings of 11th International Conference on Structural Analysis of Historical Constructions (SAHC2018), (2018) ;
(b) R. Aguilar, D. Torrealva, S. Moreira, M. Pando, L.F. Ramos (Eds.), RILEM Bookseries 18, Springer International Publishing, 2018, pp. 1839–1848, doi:http://dx.doi.org/10.1007/978-3-319-99441-3_197.

[4] P. Block, T. Ciblac, J. Ochsendorf, Real-time limit analysis of vaulted masonry buildings, Comput. Struct. 84 (2006) 1841–1852, doi:http://dx.doi.org/10.1016/j.compstruc.2006.08.002.

[5] N. Cavalagli, V. Gusella, L. Severini, Lateral loads carrying capacity and minimum thickness of circular and pointed masonry arches, Int. J. Mech. Sci. 115 (2016) 645–656.

[6] R. Livesley, Limit analysis of structures formed from rigid blocks, Int. J. Numer. Methods Eng. 12 (12) (1978) 1853–1871.

[7] J. Ochsendorf, The masonry arch on spreading supports, Struct. Eng. 84 (2) (2006) 29–35.

[8] F. Pugi, S. Galassi, Seismic analysis of masonry voussoir arches according to the Italian building code, Ing. Sismica—Ital. 30 (3) (2013) 33–55 PATRON.

[9] A. Sinopoli, M. Corradi, F. Foce, Modern formulation for preelastic theories on masonry arches, J. Eng. Mech.—ASCE 123 (3) (1997) 204–213.

[10] A. Sinopoli, M. Corradi, F. Foce, Lower and upper bound theorems for masonry arches as rigid systems with unilateral contacts, in: Sinopoli (Ed.), Arch Bridges. History, Analysis, Assessment, Maintenance and Repair, Proceedings of Arch Bridges, Balkema, Rotterdam, 1998, pp. 99–108.
[11] R. Dimitri, F. Tornabene, A parametric investigation of the seismic capacity for masonry arches and portals of different shapes, Eng. Fail. Anal. 52 (2015) 1–34.
[12] Obvis Ltd, UK, Archie-M, (1999) . http://www.obvis.com/.
[13] Obvis, Ltd, ArchieM Quick Guide, Exeter, Obvis, Ltd., 2001.
[14] M. Gilbert, Limit analysis applied to masonry arch bridges: state of the art and recent developments, Proc. of ARCH'07–5th International Conference on Arch Bridges (2007) 13–28.
[15] M. Gilbert, C. Casapulla, H.M. Ahmed, Limit analysis of masonry block structures with non-associative frictional joints using linear programming, Comput. Struct. 84 (13–14) (2006) 873–887, doi:http://dx.doi.org/10.1016/j.compstruc.2006.02.005.
[16] M. Ferris, F. Tin-Loi, Limit analysis of frictional block assemblies as a mathematical program with complementarity constraints, Int. J. Mech. Sci. 43 (1) (2001) 209–224.
[17] J. Heyman, The stone skeleton, Int. J. Solids Struct. 2 (1966) 249–279.
[18] G. Cocchetti, G. Colasante, E. Rizzi, On the analysis of minimum thickness in circular masonry arches, Appl. Mech. Rev. 64 (5) (2011)050802.
[19] S. Fernández Huerta, Mechanics of masonry vaults: the equilibrium approach, in: P. Lourenço, P. Roca (Eds.), Historical Constructions: Possibilities of Numerical and Experimental Techniques - Proceedings of the 3rd International Seminar, University of Minho, 2001.
[20] J.F. Rondeaux, A. Deschuyteneer, D. Zastavni, et al., Assessing geometrically the structural safety of masonry arches, in: Wouters, Van de Voorde, Bertels (Eds.), Building Knowledge, Constructing Histories, Proceedings of 6th International Congress on Construction History (6ICCH), 6ICCH, Brussels, Belgium, 2018, pp. 1129–1136.
[21] V. Sarhosis, S. De Santis, G. de Felice, A review of experimental investigations and assessment methods for masonry arch bridges, Struct. Infrastruct. E 12 (11) (2016) 1439–1464.
[22] Paolo Zampieri, Flora Faleschini, Mariano Angelo Zanini, Nicolò Simoncello, Collapse mechanisms of masonry arches with settled springing, Eng. Struct. 156 (2018) 363–374.
[23] F. Portioli, C. Casapulla, L. Cascini, M. D'Aniello, R. Landolfo, Limit analysis by linear programming of 3D masonry structures with associative friction laws and torsion interaction effects, Arch. Appl. Mech. 83 (2013) 1415–1438.
[24] L. Cascini, R. Gagliardo, F. Portioli, LiABlock_3D: a software tool for collapse mechanism analysis of historic masonry structures, Int. J. Archit. Herit. (2018), doi:http://dx.doi.org/10.1080/15583058.2018.1509155.
[25] F. Portioli, C. Casapulla, M. Gilbert, L. Cascini, Limit analysis of 3D masonry block structures with non-associative frictional joints using cone programming, Comput. Struct. 143 (2014) 108–121.
[26] D. O'Dwyer, Funicular analysis of masonry vaults, Comput. Struct. 73 (1) (1999) 187–197, doi:http://dx.doi.org/10.1016/S0045-7949(98)00279-X.
[27] F. Marmo, L. Rosati, Reformulation and extension of the thrust network analysis, Comput. Struct. 182 (2017) 104–118, doi:http://dx.doi.org/10.1016/j.compstruc.2016.11.016.
[28] F. Marmo, D. Masi, L. Rosati, Thrust network analysis of masonry helical staircases, Int. J. Archit. Herit. 12 (5) (2018) 828–848, doi:http://dx.doi.org/10.1080/15583058.2017.1419313.
[29] V. Alecci, G. Stipo, A. La Brusco, M. De Stefano, L. Rovero, Estimating elastic modulus of tuff and brick masonry: a comparison between on-site and laboratory tests, Constr. Build. Mater. 204 (2019) 828–838.
[30] M. Sykora, M. Holicky, Probabilistic model for masonry strength of existing structures, Eng. Mech. 17 (1) (2010) 61–70.
[31] (a) F. Pugi, Seismic analysis of masonry arch structures through the finite element model "block-joint, Proc. of COMPDYN 2013 4th ECCOMAS Thematic Conference on Computational Methods in Structural Dynamics and Earthquake Engineering, (2013) ;
(b) M. Papadrakakis, V. Papadopoulos, V. Plevris (Eds.), Institute of Structural Analysis and Antiseismic Research, National Technical University of Athens (NTUA), Greece, 2013.
[32] (a) F. Pugi, A. Francioso, Nonlinear analysis and seismic strengthening of masonry arches: the block-joint and block-block fem models, Proceedings of COMPDYN 2015-5th ECCOMAS Thematic Conference on Computational Methods in Structural Dynamics and Earthquake Engineering, (2015) ;
(b) M. Papadrakakis, V. Papadopoulos, V. Plevris (Eds.), Institute of Structural Analysis and Antiseismic Research, National Technical University of Athens (NTUA), Greece, 2015.
[33] M. Betti, G.A. Drosopoulos, G.E. Stavroulakis, Two non-linear finite element models developed for the assessment of failure of masonry arches, Comptes Rendus Mécanique 336 (2008) 1,2: 42:53.
[34] L. Giresini, M. Sassu, C. Butenweg, V. Alecci, M. De Stefano, Vault macro-element with equivalent trusses in global seismic analyses, Earthq. Struct. 12 (4) (2017) 409–423.
[35] A. Castigliano, in: A.F. Negro (Ed.), Théorie de l'equilibre des systèmes élastiques et ses applications, 1879 Torino.
[36] G. Uva, G. Salerno, Towards a multiscale analysis of periodic masonry brickwork: a FEM algorithm with damage and friction, Int. J. Solids Struct. 43 (2006) 3379–3769.
[37] G. Salerno, G. de Felice, Continuum modeling of periodic brickwork, Int. J. Solids Struct. 46 (2009) 1251–1267.
[38] E.H.F. Méry, Equilibre des voûtes en berceau, Annales des ponts et chaussées, (1840) , pp. 51–70.
[39] D.C. Drucker, Coulomb friction, plasticity and limit loads, J. Appl. Mech. 21 (1) (1954) 71–74.
[40] L. Nobile, V. Bartolomeo, Structural analysis of historical masonry arches: state-of-the-art and recent developments, Int. J. Math. Model. Meth. Appl. Sci. 9 (2015) 338–344.
[41] G. Tempesta, S. Galassi, Safety evaluation of masonry arches. A numerical procedure based on the thrust line closest to the geometrical axis, Int. J. Mech. Sci. 155 (2019) 206–221, doi:http://dx.doi.org/10.1016/j.ijmecsci.2019.02.036.
[42] S. Di Pasquale, Considerazioni Elementari sulla Statica degli Archi in Muratura, in: R.S. Olivito (Ed.), Giornata di Studio Meccanica delle Strutture Murarie, 1995, pp. 33–48 Capri (NA), Italy, 08 May1995 (in Italian).
[43] S. Galassi, G. Misseri, L. Rovero, G. Tempesta, Failure modes prediction of masonry voussoir arches on moving supports, Eng. Struct. 173 (2018) 706–717, doi:http://dx.doi.org/10.1016/j.engstruct.2018.07.015.